Acknowledgement:

* <https://hackernoon.com/14-patterns-to-ace-any-coding-interview-question-c5bb3357f6ed>

1: Sliding Window

* The Sliding Window Pattern is used to perform a required operation on a specific window size of a given array or linked list, such as finding the longest subarray contraining all 1s. Sliding windows start from the 1st element and keep shifting right by one element and adjust the length of the window accoring to the problem that you are solving. In some cases, the window size remains constant and in other cases, the sizes grows or shrinks

Diagram

Description automatically generated

Following are some ways you can identify that the given problem might require a sliding window:

* The problem input is a linear data structure such as a linked list, array, or string
* You’re asked to find the longest/shorted substring, subarray, or a desired value

Common problems you use the sliding window pattern with:

* Maxinum sum subarray of size ‘K’ (easy)
* Longest substring with ‘K’ distinct characters (medium)
* String anagrams (hard)

340

Solution Article

Approach 1: Sliding Window + HashMap

Intuition

We could take some inspiration from a simpler problem called longest subtring with at most 2 distinct characters.

To solve the problem in one pass let’s use here sliding window approach with 2 set pointers left and right serving as the windown boundaries.

The idea is to set both pointers in the position 0 and then move right pointer to the right while the window contains not more than k distinct characters.

The idea is to set both pointers in the position 0 and then move right pointer to the right while the windown contains not more than k distinct characters. If at some point, we’ve got k + 1 distinct characters, let move left pointer to keep not more than k+1 distinct characters in the window.

Table

Description automatically generated

Table

Description automatically generated

* To move sliding window along the string
* to keep not more than k distinct characters in the window
* and to update max subtring length at each step.

Next question:

* how to move the left pointer to keep only k distinct characters in the string?

Let’s use for this purpose hashmap containing all characters in the sliding window as keys and their rightmost positions as values. At each moment, this hashmap could contain not more than k + 1 elements

A picture containing table

Description automatically generated

A picture containing table

Description automatically generated

For example, using this hashmap one knows that the rightmost position of character O in “LOVELEE” window is 1 and so one has to move left pointer in the position 1+1 = 2 to exclude the character O from the sliding window.

Algorithm:

Now one could write down the algortihm

* return 0 if the string is empty or k is equal to zero
* Set both pointers in the beginning of the string left = 0 and right = 0 and init max substring length max\_len = 1
* While right pointer is less than N:
  + Add the current character s[right] in the hashmap and move right pointer to the right.
  + If hashmap contains k + 1 distinct characters, remove the left most character from the hashmap and move the left pointer so that sliding window contains again k distinct characters only
  + Update max\_len

Complexity Analysis

Do we have here the best possible time complextiy O(N) as it was for more simple problem with at most 2 distint characters?

For the best case when input string contains not more than k distinct characters the answer is yes. It’s only one pass along the string with N characters and the time complexity is O(N).

For the worst case when the input string contains n distinct characters, the answer is no.

Abcdef => N distinct characters.

In that case, at each step one uses O(k) time to find a minimum value in the hashmap with k elements and so the overall time complexity is O(Nk).

* Time complexity: O(N) in the best case of k distinct characters in the string and O(Nk) in the worst case of N distinct characters in the string.
* Space complexity: O(k) since additional space is used only for a hashmap with at most k + 1 elements.

Not more than k distinct characters:

Aaabbcccc k = 2

Takeaways:

- need a hashMap of key/value: character: last seen position

- slide window.

- move the left pointer after deleting a character: left = leftMost + 1

* Look at the solution for improvement

Graphical user interface, text, application

Description automatically generated

Don’t store the index, instead, store the count of each character, once right point (count == 0), we increase the number of distinct character by 1, left pointer keeps once count reaches 0, we decrease it.

Kadane’s Algorithm

Kadane’s original algorithm solves the problem version when empty subarrays are admitted. It scans the given array A[1…n] from left to right. In the jth step, it computes the subarray with the largest sum ending at j; this sum is maintained in variable current\_sum. Moreover, it computes the subarray with the largest sum anywhere in A[1…j], maintained in variable best\_sum, and easily obtained as the maxinum of all values of current\_sum seen so far, cf. line 7 of the algorithm.

As a loop invariant, in the jth step, the old value of current\_sum holds the maximum over all i in {1, …., j} of the sum A[i] + … + A[j-1]. Therefore, current\_sum + A[j] is the maximum over all i in {1, ….j} of the sum A[i] + … + A[j]. To extend the latter maxinm to cover also the case i = j + 1, it is sufficient to consider also the empty subarray A[j + 1 … j]. This is done in line 6 by assigning max (0, current\_sum + A[j]) as the new value of current\_sum, which after that holds the maxinum over all i in {1, …, j+1} of the sum A[i] + … + A[j]

For the variant of the problem which disallows empty subarrays, best\_sum should be initialized to negative infinity instead and also in the for loop current\_sum should be updates as max(x, current\_sum + x). In that case, if the input contains no positive element, the return value is that of the largest element (i.e, the value closet to 0), or negative infinity if the input was empty.

maxSubArray with Kadane’s Algorithm

* Initialize 2 integer variables. Set both of them equal to the first value in the array.
  + currentSubarray will keep the running count of the current subarray we are focusing on
  + maxSubarray will be our final return value. Continously update it whenever we find a bigger subarray
* Iterate through the array, starting with the 2nd element (as we used the first element to initialize our variable).
  + For each number, add it to the currentSubarray we are building. Remember to update maxSubarray everytime we find a new maxinum
* Return maxSubarray.

**Divide and Conquer vs DP: references.**

**https://pediaa.com/what-is-the-difference-between-divide-and-conquer-and-dynamic-programming/**

* Divide and conquer: divide the problem to sub problems and then combines the solutions of the sub-problems to obtain the solution of the main problem.
  + Sub problems are independent of each other
* DP: divide the problem to sub problems, and uses the result of the sub problems to find the optimum solution of the main problem.
  + Sub problems are interdepent of each other.
  + Dynamic programming does not solve the subproblems independently
    - It stores the answers of subproblems to use them for similar problems.
  + Usually use for optimization

|  |  |
| --- | --- |
| Divide and Conquer | DP |
| An algorithm that recursively breaks down a problem into 2 or more sub-problems of the same or related type until it becomes simple enough to be solved directly | An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property |
| Sub problems are independent of each other | Subproblems are interdependent |
| Recursive | Non-recursive |
| More time-consuming as it solves each sub problem independently | Less time-consuming as it uses the answers of the previous subproblems. |
| Less efficient | More efficient |
| Used by merge sort, quicksort, and binary search | Used by matrix chain multiplication, optimal binary search tree. |

**What is Divide and Conquer**

Divide and conquer divides the main problem into small subproblems. The subproblems are divided again and again. At one point, there will be a stage where we cannot divide the subproblems further.

Then, we can solve each sub problem independtly. Finally, we can combine the solutions of all subproblems to get the solution to the main problem.

Diagram, schematic

Description automatically generated

Diagram

Description automatically generated

There are three main steps in divide and conquer. They are as follows

**Divide (break**) – involves splitting the main problem into a collection of subproblems

**Conquer (solve)** – involves solving each sub problem separately.

**Combine (merge)** – joins the solutions of the sub problems to obtain the solution of the main problem.

**What is Dynamic Programming.**

Dynamic Programing divides the main problem into smaller subproblems, but it does not solve the subproblems independently. It stores the results of the sub problems to use when solving similar subproblems. Storing the results of subproblems is called memorization. Before solving the current subproblem, it checks the results of the previous subproblems. Finally, it checks the results of all sub problems to find the best solution or the optimal solution. This method is effective as it does not compute the answers again and again. Usually, dynamic programming is used for optimization.

.Diagram, shape

Description automatically generated

Elements of dynamic programming are as follows.

* **Simple subproblems**: divide the original problem into small subproblems that have a similar structure.
  + To think about more
* **Optimal substructure of the problem**: the optimal solution to the main problem is within the optimal solution to its sub problems.
  + Longest subarray with k distinct character
* **Overlapping subproblems**: situation of solving the same sub problems again and again.
  + Fibonoci sequence with memoization.

**Difference Between Divide and Conquer and Dynamic Programming.**

***Definition.***

Divide and conquer is an algorithm that recursively breaks down a problem into 2 or more sub problems of the same or related type until it becomes simple enough to be solved directly. However, DP is an algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substrucutre property.

***Type.***

The main difference between divide and conquer and dynamic programming is that divide and conquer is recursive while DP is non-recursive.

***Sub Problems.***

In divide and conquer, the subproblems are independent of each other. However, in dynamic programming, the subproblems are interdependent. Hence, this is another major difference between divide and conquer and DP.

**Time Consumption**

Time consumption is another difference between divide and conquer and dynamic programming. Divide and conquer solves each subproblem independently. Therefore, it is more time-consuming. DP on the other hand, uses the answers of the previous sub problems. Thus, it is less time-consuming.

**Efficiency**.

Efficiency also makes a difference between divide and conquer and dynamic programming. Dynamic programming is more efficient than divide and conquer.

**Applications**

Merge sort, quicksort, and binary search use divide and conquer while matrix chain multiplication and optimal binary search tree use DP.

**Conclusion**

The main difference between divide and conquer and dynamic programming is that the divide and conquer combines the solutions of the subproblems to obtain the solution of the main problem while DP uses the result of the subproblems to find the optimum solution of the main problem.

If we were to split our input in haft, then logically the optimal subarray either:

* Uses elements only from the left side
* Uses elements only from the right side
* Uses a combination of elements from both left and right side

Thus, the answer is simply the largest of:

* The maximum sub array contained only in the left side
* The maximum sub array contained only in the right side
* The maximum sub array that can use elements from both sides

Finding the maxinum subarray from the left and right haft is straightforward – just recurse using the respective haft of the array. So, the hard part is figuring out how to find the best subarray that uses elements from both sides. Let’s use smaller example, nums = [5, -2, 1, -3, 4, -2, 1].

Since we want to use elements from both sides, we also must use the middle element, -3. Now, we can also take from the left and the right, but every element must be connected to the middle (since we’re still forming a subarray).

The fact that every element must be connected to the middle actually makes our lives a lot easier – every subarray we consider must contain the element immediately beside the center, which means there are only as many subarrays as here are elements. In our example, the right side is [4, -2, 1]. There are only 3 subarrays to consider [4], [4, -2] and [4, -2, 1]. To find the best chain of elements we can take from a haft, iterate from the middle to the end (start of the array for the left haft) and continuously update some counter curr.

Graphical user interface, application

Description automatically generated

A picture containing funnel chart

Description automatically generated Graphical user interface

Description automatically generated with medium confidence

Chart

Description automatically generated

Algo:

Now that we know how to find the best subarray containing elements from both sides of any given array, the algo is as follows:

1: Define a helper funtion that we will use for recursion.

* This function will take an input left and right which defines the bounds of the array. The return value of this function will be the best possible subarray for the array that fits between left and right.
* If left > right, we have an empty array. Return negative infinity
* Find the midpoint of our array. This is (left + right) / 2, rounded down. Using this midpoint, find the best possible subarray that uses elements from both sides of the array.
* The best subarray using elements from both sides is only 1 of 3 possibilities. We still need to find the best subarray using only the left or right halves. So call this function again, once with the left haft, and once with the right half.
* Return the largest of the 3 values – the best left haft sum, the best right haft sum, and the best combined sum.

2: Call our helper funtion with the entire input array (left = 0, right = length – 1). This is our final answer, so return it.

/\*

# base case – empty array

# iterate from the middle to the beginning

# reset curr and iterate from the middle to the end

# the best\_combined\_sum uses the middl element and

# the best possible sum from each half.

# find the best subarray possible from both halves

# the largest of the 3 is the answer for any given input array.

# our helper funtion is designed to solve this problem for

# any array – so just call it using the enture input!

\*/